An Introduction to Computational Complexity

Michael Soltys

An Introduction to Computational Complexity

WYDAWNICTWO UNIWERSYTETU JAGIELLOŃSKIEGO

Book Series: Mathematics

The publication of this volume was supported by the Jagiellonian University
– the Theoretical Computer Science Department

EDITOR IN CHIEF
Wojciech Słomczyński

REVIEWER
Paweł M. Idziak

TECHNICAL EDITOR
Grzegorz Gutowski

BOOK SERIES COVER DESIGNER
Andrzej Harasz

COVER DESIGNER
Bartłomiej Drosdziok

EDITOR AND PROOFREADER
Jerzy Hrycyk

ISBN 978-83-233-2864-3

www.wuj.pl

Wydawnictwo Uniwersytetu Jagiellońskiego
Editorial Offices: ul. Michałowskiego 9/2, 31-126 Kraków
phone: 012-631-18-81, 012-631-18-82, phone/fax: 012-631-18-83
Sales: ul. Wrocławska 53, 30-011 Kraków
phone: 012-631-01-97, phone/fax: 012-631-01-98
phone: 0506-006-674, e-mail: sprzedaz@wuj.pl
bank: PEKAO SA O/Kraków, IBAN PL80 1240 4722 1111 0000 4856 3325

To my parents

Contents

Preface

Complexity theory asks what makes certain problems computationally difficult. It also investigates the relationship between computability and provability; indeed, the famous P versus NP question can be seen in this light: if all solutions to the instances of a given problem can be verified efficiently, then can those solutions be also computed efficiently? The quest of complexity is to assess the computational difficulty of a given problem; for example, how hard is it to determine if a graph can be colored with three colors so that no two adjacent nodes are of the same color?

Complexity upper bounds provide an algorithm that solves the problem in the most efficient way—with respect to resources such as time or space or circuit size. The lower bounds claim that the problem cannot be solved within some quantity of those resources. It is not surprising that upper bounds are easier—we are adept at finding quick algorithms—while good lower bounds are usually very difficult to achieve. Simply put, we are not very proficient at showing that *all* algorithms in a given complexity class are not up to a certain task. After all, lower bounds require a lot of ingenuity in order to show that no ingenuity whatsoever can solve a problem within a given bound on resources. Expressed in another way, existential statements seem more amenable to proofs than universal ones.

The intended audience for this book are graduate students in computer science and mathematics who want to quickly familiarize themselves with computational complexity. As such, this book aims more at depth than breadth. There are many excellent comprehensive guides to computational complexity (in particular classics such as [**Pap94**] and [**Sip06**]). Here, on the other hand, the reader will find major results of complexity presented with a minimum of background.

The highlights are as follows. In chapter 1 we present two applications of the crossing sequences method: we show that a single tape Turing machine requires $\Omega(n^2)$ many steps to decide the language of palindromes (which can also be seen as an application of rudimentary Kolmogorov complexity), and that languages decidable with $o(\log \log n)$ space are in fact regular.

In chapter 2 we use the self-reducibility of Sat to show that tally sets cannot be NP-complete unless $\mathsf{P} = \mathsf{NP}$, and we prove the Karp-Lipton theorem (if $\mathsf{NP} \subseteq \mathsf{P/poly}$, then PH collapses to its second level). We also introduce the so called "padding technique," and use it to prove Ladner's theorem (if $\mathsf{P} \neq \mathsf{NP}$, then there are languages in $\mathsf{NP} - \mathsf{P}$ which are not NP-complete).

In chapter 3 we deal with space complexity, and present Savitch's theorem as well as the inductive counting technique, and prove the Immerman-Szelepcsényi theorem

(i.e., NL is closed under complementation). Chapter 3 ends with interactive proof systems and a proof of IP = PSPACE.

In chapter 4 the Hierarchy theorems are presented, and we prove the somewhat technical result stating that "$\mathsf{P}^A \neq \mathsf{NP}^A$ with probability 1" (with respect to a random oracle A). The Polytime Hierarchy is examined in detail, and we prove some properties of Alternating Turing machines. We end chapter 4 with an application of the so called "Bennett's trick": Nepomnjascij's theorem.

In chapter 5 we have Shannon's (easy) lower bound for circuits, and we introduce the probabilistic method in order to give two different proofs showing that PARITY is not in AC^0. We end the chapter with a characterization of nonuniform computation via Turing machines with advice tapes.

In chapter 6 we prove Haken's lower bound for the size of resolution refutations of the pigeonhole principle. We also give a lower bound for resolution based on the idea of interpolation and Razborov's lower bound for monotone circuits computing CLIQUE.

The last chapter, chapter 7, starts with three examples of randomized algorithms, we introduce the notion of amplification, and present Sipser's theorem ($\mathsf{BPP} \subseteq \Sigma_2^p$) and finish with Toda's theorem ($\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$).

In the Appendix (chapter 8) we collect miscellanea: a section with different NP-complete problems; some number theory (mostly background for the Rabin-Miller algorithm—algorithm 7.2); a section on the RSA public key encryption; the Isolation Lemma (used in the proof of Toda's theorem); and a section on Berkowitz's algorithm (an NC^2 algorithm for computing the determinant of a matrix).

Many standard results, not mentioned in the above outline, are sprinkled throughout the text. On the other hand, there are many complexity classes that do not make an appearance. The tapestry of complexity classes is truly overwhelming; *"But the man who sets himself the task of singling out the thread of order from the tapestry will by the decision alone have taken charge of the world"*.[1]

This book contains one semester worth of material, that is, it is intended for a ten to twelve week course, that meets for two to three lecture hours a week.

Notation

The symbol Σ will have several (but all standard) meanings. It will denote a finite alphabet of symbols, such as the binary alphabet $\Sigma = \{0, 1\}$, and Σ^* denotes the set of all finite strings over the alphabet Σ (i.e., Kleene's[2] star of Σ). We shall use Σ_i to denote alternations; for example, Σ_i^p denotes the set of relations expressible by a polytime predicate with i alternations of quantifiers in front of it, where the first quantifier is $\exists$ (see §4.3).

$\vec{x} = x_1, x_2, \ldots, x_n$, i.e., a vector of variables, while $\bar{x}$ denotes the same as $\neg x$, i.e., a negation of a Boolean variable. We use $[n]$ to denote the subset of natural numbers ($\mathbb{N}$) consisting of $\{1, 2, \ldots, n\}$. We use ":=" to make definitions.

---

[1] Cormac McCarthy, "Blood Meridian Or the Evening Redness in the West", Vintage Books, first Vintage edition, May 1992, pg. 199.

[2] After Stephen Cole Kleene. Σ^i often denotes the set of all strings of length i over the alphabet Σ, so Σ^* can be defined as $\bigcup_{i=0}^{\infty} \Sigma^i$, where $\Sigma^0 = \{\varepsilon\}$, and where ε is the (unique) string of length 0.

We use $A \subseteq B$ to denote that A is a subset of B, and possibly $A = B$. We use $A \subset B$ to denote that A is a *proper* subset of B, i.e., $A \subset B$ iff $A \subseteq B$ and $A \neq B$. Also, $A - B$ will denote set difference: all those elements in A which are not in B. We use $\wedge, \vee, \neg, \oplus, \rightarrow, \leftrightarrow$ to denote Boolean and, or, not, xor (exclusive or), implies, equivalent, respectively. Sometime we use the more expressive names, `And`, `Or`, `Not`, `Xor`. We use `T`, `F` (as well as $1, 0$) to denote the Boolean constants true and false, respectively, and we use $\Gamma \vDash \alpha$ to denote that the set of formulas Γ logically implies the formula α. We sometimes use $t \vDash \alpha$ to say that the truth assignment t satisfies the formula α; thus "$\vDash$" has a dual meaning.

We use the standard *big-Oh notation*, $g(n) \in O(f(n))$ if there exist constants c, n_0 such that for all $n \geq n_0$, $g(n) \leq cf(n)$, and the *little-oh* notation, $g(n) \in o(f(n))$, which denotes that $\lim_{n\to\infty} g(n)/f(n) = 0$. We also say that $g(n) \in \Omega(f(n))$ if there exist constants c, n_0 such that for all $n \geq n_0$, $g(n) \geq cf(n)$. Finally, we say that $g(n) \in \Theta(f(n))$ if it is the case that both $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$.

We shall need a little bit of number theory: let $\mathbb{Z}_n := \{0, 1, 2, \ldots, (n-1)\}$, the set of numbers modulo n, and let $\mathbb{Z}_n^*$ denote the subset of $\mathbb{Z}_n$ consisting of those numbers co-prime with n. We shall write $x \equiv_n y$ and $x \equiv y \pmod{n}$ to denote that $n|(x-y)$, i.e., that n divides $(x-y)$. Let $(n)_b$ be the binary representation of the integer n; for example $(5)_b = 101$.

Our logarithm function, $\log x$, is assumed to be in base 2, i.e., $\log x := \log_2 x$. Using standard notation, we let $\ln x := \log_e x$.

Sans-serif fonts will be used to denote complexity classes, for example NP, and SMALL CAPS FONTS will be used to denote languages (i.e., problems), for example MINFORMULA.

Acknowledgments

First of all, it will be clear to anyone reading this book that it relies heavily on many other books and papers (each chapter ends with a Notes section that points the reader to the appropriate sources). I am greatly indebted to all those authors.

This book came about as the result of the author teaching a graduate complexity course at McMaster University (Hamilton, Canada) in the years 2002–2006; during a visit at the Algorithmics Research Group of the Jagiellonian University (Kraków, Poland) in the summer 2007; teaching an *Ulam Seminar* at the University of Colorado at Boulder in the Fall 2007; and in February 2008, during the IX Escuela de Verano de Ciencias Informáticas, Universidad Nacional de Río Cuarto, Argentina.

I am grateful to all the students who attended these courses, and in particular, I am deeply grateful to the following proof readers: Lech Duraj, Grzegorz Gutowski, Grzegorz Herman (for reading the *entire* book), Jan Jeżabek, Ryan Lortie, Bartosz Walczak, and Craig Wilson.

I am very grateful to Prof. Jan Mycielski for comments and corrections, especially for a very careful proofreading of §7.1.2, and to Prof. Hugo Ryckeboer.

Finally, I would like to express my special thanks to Prof. Paweł Idziak for his help in publishing this book, and to Grzegorz Gutowski and Dai Tri Man Lê for a careful proof reading of the final manuscript.

1

Turing machines

1.1. Definition

A *Turing machine*[1] (TM) is a tuple $(Q, \Sigma, \Gamma, \delta)$ where Q is a finite set of *states* (always including the three special states q_{init}, q_{accept} and q_{reject}), Σ is a finite *input alphabet* (and unless it is otherwise specified it is $\{0, 1\}$), Γ is a finite *tape alphabet*, and it is always the case that $\Sigma \subseteq \Gamma$ (it is convenient to have symbols on the tape which are never part of the input),

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{\text{Left}, \text{Right}\}$$

is the *transition function* which says that when the TM is in state q and the head is reading a symbol σ, it changes to a new state q', overwrites σ with a new symbol σ', and then the head moves left or right. Note that neither q' nor σ' have to be different from q and σ, respectively. The definition of a TM can be adapted to suit any occasion: there can be more than one tape, and the tape(s) can be infinite in one or two directions, etc. From a computational point of view, all these models are equivalent.

A *configuration* is a tuple (q, w, u) where $q \in Q$ is a state, and where $w, u \in \Gamma^*$, the cursor is on the last symbol of w, and u is the string to the right of w. A configuration (q, w, u) *yields* (q', w', u') in one step, denoted as $(q, w, u) \overset{M}{\to} (q', w', u')$ if one step of M on (q, w, u) results in (q', w', u'). Analogously, we define $\overset{M^k}{\to}$, yields in k steps, and $\overset{M^*}{\to}$, yields in any number of steps, including zero steps. The initial configuration, C_{init}, is $(q_{\text{init}}, \triangleright, x)$ where q_{init} is the initial state, x is the input, and $\triangleright$ is the left-most tape symbol, which is always there to indicate the left-end of the tape.[2] For a TM with k tapes, a configuration is given by $(q, w_1, u_1, \ldots, w_k, u_k)$.

Given a string w as input, we "turn on" the TM in the initial configuration C_{init}, and the machine moves from configuration to configuration. The computation ends when either the state q_{accept} is entered, in which case we say that the TM *accepts* w, or the state q_{reject} is entered, in which case we say that the TM *rejects* w. It is

---

[1] Alan Turing in 1936 ([**Tur37**]) was the first to use imaginary computers (which came to be known as *Turing machines*) for characterizing the class of algorithmic functions.

[2] Note that $\triangleright$ is not part of the alphabet Σ, but is part of the *tape alphabet* Γ which always contains Σ. The symbol $\triangleright$ is convenient for denoting the left-end of the tape—and it can be moved to the right, when it is useful to ignore some initial segment of the tape; we follow [**Pap94**, definition 2.1], where, besides the left-end of tape symbol $\triangleright$, the blank square $\sqcup$ is also introduced as part of Γ and not part of Σ. The point is that we can define Turing machines in a convenient way, with small alterations at will.

possible for the TM to never enter q_{accept} or q_{reject}, in which case the computation does not halt.

Given a TM M we define $L(M)$ to be the set of strings accepted by M, i.e., $L(M) = \{x | M \text{ accepts } x\}$, or, put another way, $L(M)$ is the set of precisely those strings x for which $(q_{\text{init}}, \triangleright, x)$ yields an accepting configuration.

EXERCISE 1.1. Give a formal definition of $L(M)$ using $\overset{M^k}{\to}$ and one using $\overset{M^*}{\to}$.

Alan Turing showed the existence of a so called *Universal Turing machine* (UTM); a UTM is capable of simulating any TM from its description. A UTM is what we mean by a computer, capable of running any algorithm. The proof is not difficult, but it requires care in defining a consistent way of presenting TMs and inputs.

EXERCISE 1.2. Convince yourself that one can construct a UTM.

There are many definitions of computation other than the one given by Alan Turing; for example the so called *Unlimited Register Ideal Machine* (URIM). A URIM program P is a sequence of commands $\langle c_1, c_2, \ldots, c_n \rangle$, operating on a finite (but arbitrarily large) set of registers $R_1, R_2, \ldots, R_m$ (which contain natural numbers in, say, unary notation), and each command is one of the following three types:

$$R_i \leftarrow 0,$$
$$R_i \leftarrow R_i + 1,$$
$$\texttt{goto } c_i \texttt{ if } R_j = R_k.$$

In the last command, if $R_j \neq R_k$, then the next command on the list is run. If we run out of commands (or are sent to an $i > n$ by a `goto`) then the program terminates. We can always assume that the input is given in R_1 at the beginning, and the output is given in R_m at the end (with 0 being "no" and 1 being "yes," if we are computing a decision problem and a number if we are computing a function). Note that the subscripts i, j, k are part of the instruction, i.e., they are not variable, but rather "hard-coded" into the program.

Yet another model of computation is given by *recursive functions*. A function f ($f(\vec{x})$, where $\vec{x} = x_1, x_2, \ldots, x_n$) is defined by *composition* from $g, h_1, \ldots, h_m$ if $f(\vec{x}) = g(h_1(\vec{x}), \ldots, h_m(\vec{x}))$, and f is defined by *primitive recursion* from g, h if

$$f(\vec{x}, 0) = g(\vec{x}),$$
$$f(\vec{x}, y+1) = h(\vec{x}, y, f(\vec{x}, y)).$$

There are three *initial functions* $Z, S, I_{n,i}$, where Z is the constant function equal to 0, $S(x) = x + 1$ (the successor function), and $I_{n,i}(\vec{x}) = x_i$ (the projection function). We say that a function is *primitive recursive* if it can be obtained from the initial functions by finitely many applications of primitive recursion and composition. Finally, we say that a function f is defined by *minimization* from g if $f(\vec{x}) = \mu y[g(\vec{x}, y) = 0]$, where $\mu y[g(\vec{x}, y) = 0]$ is the smallest b such that $g(\vec{x}, b) = 0$, and a function is *recursive* if it can be obtained from the initial functions by finitely many applications of primitive recursion, composition, and minimization.[3] Note that primitive recursive functions

[3]It turns out that only one application of minimization is sufficient.

are always total, but recursive functions are not necessarily so—there may be an $\vec{x}$ for which there is no b such that $g(\vec{x}, b) = 0$.

EXERCISE 1.3. Convince yourself that TMs, URIMs, and recursive functions, are all equivalent models of computation.

1.2. Basic properties

We define $\mathsf{TIME}(f(n))$ to be the class of languages decidable by TMs running in time $O(f(n))$ (i.e., TMs that take at most $O(f(n))$ many steps, on any input of size n, before making a decision). $\mathsf{SPACE}(f(n))$ requires a little bit more care because we want to make sense of sub-linear space (i.e., less working space than the actual input size; for example, logarithmic space). So for space we assume that we have a read-only input tape on which the input string is presented, and a work tape on which we bound how much tape we are allowed to use. Thus, we say that $\mathsf{SPACE}(f(n))$ is the class of languages decidable by TMs that use at most $O(f(n))$ squares of the work tape.

THEOREM 1.4. Given a k-tape TM M operating within time $f(n)$, we can construct a single-tape TM M' operating within time $O(f(n)^2)$, such that $L(M) = L(M')$.

EXERCISE 1.5. Prove theorem 1.4.

THEOREM 1.6 (Speed-Up). Suppose that a TM decides a language L in time bounded by $f(n)$. For any $\varepsilon > 0$, there exists a TM that decides L in time $f'(n) = \varepsilon \cdot f(n) + n + 2$.

EXERCISE 1.7. Prove theorem 1.6.

EXERCISE 1.8. What does theorem 1.6 say about languages decidable in time $O(n)$? In general, what does this theorem say about the "big-Oh" notation?

EXERCISE 1.9. Suppose that we insist that the tape alphabet be $\{\triangleright, \sqcup, 0, 1\}$. Can the speed-up theorem be still applied?

In a nondeterministic TM the transition function δ becomes a transition relation Δ, so "yields" is now a relation as well. A nondeterministic computation can be viewed as a tree, where the nodes are configurations, and we branch on all the possible choices allowed by Δ. The machine accepts if at least one branch ends in an accepting configuration. Given a TM M, let d be its *degree of nondeterminism*, meaning that d bounds the number of choices of Δ (this number is a constant, independent of the input, and $d = 1$ for deterministic machines). Formally, $d := \max_{q \in Q, \sigma \in \Gamma} |\Delta(q, \sigma)|$.

We define $\mathsf{NTIME}(f(n))$ to be the class of languages decidable by nondeterministic TMs where each branch is of length bounded by $f(n)$. We define $\mathsf{NSPACE}(f(n))$ to be the class of languages decidable by nondeterministic TMs where the space used on the work-tapes is bounded by $f(n)$ and where the length of each branch is bounded by the number of configurations possible with space $f(n)$.

THEOREM 1.10. $\mathsf{NTIME}(f(n)) \subseteq \bigcup_{c>1} \mathsf{TIME}(c^{f(n)})$.

PROOF. Let $L \in \mathsf{NTIME}(f(n))$, so $L = L(M)$ for some nondeterministic TM that runs in time $f(n)$. Let d be the degree of nondeterminism of M. The computation tree has size at most $d^{f(n)}$, and the tree can be traversed (in a breadth-first manner) with a pointer to a location in the tree. This pointer can be encoded with a number in base d of length $f(n)$. The traversal takes time $O(d^{f(n)})$. □

1.3. Crossing sequences

1.3.1. A lower bound for palindromes. If $x = x_1 x_2 \dots x_n \in \Sigma^*$ is a string, then the *reverse* of x, denoted x^R, is just $x_n \dots x_2 x_1$. We say that a string is a *palindrome* if $x = x^R$, and we define the language of palindromes to be the set $L_{\text{pal}} = \{x \in \{0,1\}^* | x = x^R\}$.

THEOREM 1.11. Suppose that M is a one-tape TM that decides the language L_{pal}. Then, M requires $\Omega(n^2)$ many steps.

PROOF. We first define the *i-th crossing sequence of M on x* to be

$$\{(q_1, \sigma_1), (q_2, \sigma_2), \dots, (q_m, \sigma_m)\},$$

and it means that the first time M crosses from square i to square $i+1$, M leaves σ_1 on the i-th square, and finds itself in state q_1, and then the first time it crosses from square $i+1$ to square i, it arrives at square i in state q_2 and encounters σ_2 on the i-th square, etc.

Note that the odd pairs denote crossings left-to-right, and even pairs denote crossings right-to-left, and that $\sigma_{2i} = \sigma_{2i-1}$.

Now consider inputs of the form $x0^n x^R$, where $|x| = n$. Let $T_M(x)$ be the number of steps that M takes to decide (and accept) $x0^n x^R$.

There must be some i, $n < i \leq 2n$, i.e., some square in 0^n, for which the i-th crossing sequence has length $m \leq \frac{T_M(x)}{n}$. The reason is that the sum of the length of the crossing sequences corresponding to each square in 0^n has to be bounded above by $T_M(x)$, and therefore not all squares can have crossing sequences that are "long" ($> \frac{T_M(x)}{n}$). Let S be this "short" ($\leq \frac{T_M(x)}{n}$) crossing sequence. (Note that we assumed in this analysis that the TM never stays put; it always either moves right or left. But this is not a crucial restriction as we can always "speed-up" a TM, combining several moves into one, so that it never stays put. Besides, in our official definition, TMs always move right or left.)

CLAIM 1.12. M, n, i, S describe x uniquely.

PROOF. To see this, we show how to uniquely "extract" x from M, n, i, S. For each $x \in \{0,1\}^n$, we simulate M on $x0^i$. The first time M wants to cross from square i to square $i+1$, we check that it would have done so with the pair (q_1, σ_1), and we immediately reset the state to q_2 and put the head back on square i and continue. The second time M crosses from square i to square $i+1$ we again check that it does so with (q_3, σ_3), and again we immediately reset the state to q_4 and put the head back on square i and continue. If all the odd-numbered crossing pairs are correct

(i.e., correspond to those in S), we know that we have found x. Otherwise, we move on to examine the next x.

Our procedure identifies x correctly: suppose that some $y \neq x$ passed all the tests. Then M would have accepted $y0^n x^R$ which is *not* a palindrome. This would contradict the correctness of M.

We must modify M to return to $\triangleright$ before accepting, to ensure an even-length crossing sequence, but this adds only a linear number of steps to the computation. □

So M, n, i, S describe x uniquely, and they can be encoded with

$$c_M + \log n + \log n + c \cdot m \leq c_M + 2\log n + c \cdot \frac{T_M(x)}{n}$$

many bits. However, for every n, no matter how we encode strings, there must be a string x_0 whose encoding requires at least n many bits (otherwise, we would have a bijection from the set of strings of length n to the set of strings of length $n-1$, which is not possible).

Therefore, we have that

$$n \leq c_M + 2\log n + c \cdot \frac{T_M(x_0)}{n}, \tag{1}$$

which gives us the result. □

Exercise 1.13. Theorem 1.11 says that any single tape TM M deciding L_{pal} requires $\Omega(n^2)$ steps. This means that there exist constants $b, n_0 \in \mathbb{N} - \{0\}$ such that for any $n \geq n_0$, there exists an $x \in \{0,1\}^n$ such that M takes at least $\frac{1}{b}n^2$ many steps to decide x. Can we really make the step from the conclusion of the above proof to the statement of theorem 1.11?

We say that a model of computation is *robust* if it is insensitive to "small" changes in the definition. This is a vague notion, but there are good examples of robust models: TMs with one or several tapes, infinite in one or two directions, etc., all capture the set of recursive languages; if we restrict these TMs to be polytime, they are still insensitive to these modifications, so polytime TMs are a robust model of computation as well.

Let LT be the class of languages decidable in linear time, i.e., in time $O(n)$. Formally, $\mathsf{LT} = \mathsf{TIME}(n)$; we are going to encounter this class again in §4.5. From theorem 1.11 we know that LT is *not* robust because L_{pal} can be decided in $O(n)$ steps on a two-tape TM, while on a single-tape TM it requires $\Omega(n^2)$ steps. However, NLT (nondeterministic linear time) is quite robust as the next lemma shows.

Lemma 1.14. If M_k^{nlt} is a nondeterministic linear time bounded TM with k tapes, then M_k^{nlt} can be simulated by M_2^{nlt}.

Proof. Let us assume that the input to M_2^{nlt} is written on tape 1. M_2^{nlt} starts by guessing the entire computation of M_k^{nlt}, and writing the guess on tape 2. This computation does not contain the tape configurations of M_k^{nlt}, but rather for each step of M_k^{nlt} it writes down the (supposed) state and symbols scanned by each of the k heads. Note that this information (if correct) is enough to determine the next move of M_k^{nlt}.

Now M_2^{nlt} checks that the computation it has written is correct, as follows. First it checks that the state sequence is consistent, by making one pass on tape 2, and checking that at each step the next state is what it should be, given the information from the previous step. Next, for each tape i of M_k^{nlt}, it makes a pass on tape 2 and checks that the scanned symbol information it has written for tape i is consistent. It does this by using tape 1 to simulate tape i, and using the information it has written on tape 2 concerning what the other $k-1$ tapes are scanning.

If all these consistency checks are passed, then the information on tape 2 is correct. Hence M_2^{nlt} can figure out what M_k^{nlt} did. □

EXERCISE 1.15. In the above proof, show formally by induction that the t-th step is simulated correctly.

EXERCISE 1.16. Show that L_{pal} is in $\mathsf{TIME}(n) \cap \mathsf{SPACE}(\log n)$.

1.3.2. Little space is no space at all. A language is *regular* if it can be decided by a Deterministic Finite Automaton (DFA). A DFA is just a TM with no tapes besides the input tape, where it scans the input from left-to-right, changing states as it reads the input, but not writing anything—and in particular never turning back, and it enters an accepting or rejecting state immediately after it scanned the last symbol of the input. See [**Sip06**] for the background on DFAs.

In this section we show that if a language can be decided with $o(\log\log n)$ space,[4] then the language is in fact regular.

THEOREM 1.17. If a language L can be decided by a TM M such that M has a read-only input tape, and a work-tape where space is bounded by a function in $o(\log\log n)$, then L is regular.

The next exercise shows that $\log\log n$ is an exact threshold.

EXERCISE 1.18. Consider the language L over $\{0, 1, \#\}$ given by

$$L = \{\#b_k(0)\#b_k(1)\#b_k(2)\#\ldots\#b_k(2^k-1)\#|k \geq 0\},$$

where $b_k(i)$ is the k-bit binary representation of $i \leq 2^k - 1$. Show that this language is (i) not regular, and (ii) decidable in space $O(\log\log n)$.

We show first that if a language is decidable by a TM with work-tape space bounded by a constant, then the language is regular. Next, we show that if a language is not regular, then the work-tape requires $\Omega_{\text{weak}}(\log\log n)$ space. Note that, as the notation indicates, Ω_{weak} is the "weak" version of Ω, which means that for some constant c, $c\log\log n$ is a lower bound for *infinitely many* n's. On the other hand, the standard version of Ω would assure a lower bound for *all* n (sufficiently big), rather than infinitely many. Confer with exercise 1.13.

We assume that our TMs have a read-only input tape, and a work-tape that is read `&` write, and we bound the space on the work-tape.

CLAIM 1.19. If L can be decided in space bounded by a constant (i.e., there is a constant k bounding the number of work-tape-squares that the TM is allowed to visit during any computation), then L is in fact regular.

[4]Recall that $o(f(n))$ is "little-oh" of $f(n)$, and $g(n) \in o(f(n))$ if $\lim_{n\to\infty} g(n)/f(n) = 0$.

PROOF. Note that we can get rid of constant space by keeping a record of all the possible $k \cdot |\Gamma|^k$ configurations (constantly many) by encoding them as states: there are k positions for the work-tape head, and the size of the alphabet is $|\Gamma|$, so $|\Gamma|^k$ are the possible contents of the work-tape. But this is not enough, because the head on the input-tape may move back-and-forth, so we cannot conclude directly that the machine can be simulated by a finite automaton.

We need to show that the machine can be transformed so that the head on the input-tape moves only from left-to-right. This can be accomplished as follows: for any string $s \in \Gamma^*$, define two functions $f_s, g_s : Q \longrightarrow Q$, where $f_s(q_1) = q_2$ if when the machine is started on s in state q_1, with the head on the first symbol of s, then the first time the machine leaves s (by moving from the last symbol of s to the empty square $\sqcup$) or halts, it does so in state q_2. Let g_s be defined similarly, but instead of the head starting on the first symbol of s, it starts on the last symbol of s. Note that there are $|Q|^{|Q|}$ many functions from Q to Q, so they can all be hard-wired into the machine. We can now make the machine move only from left-to-right, and accept or reject after reading the last symbol of s, by doing the following: when we leave the i-th symbol of s, we know the values of $f_{s_1 s_2 \ldots s_i}$ and $g_{s_1 s_2 \ldots s_i}$. We want to compute $f_{s_1 s_2 \ldots s_{i+1}}$ and $g_{s_1 s_2 \ldots s_{i+1}}$. This is easy using the transition function of the original machine.

It is crucial that the two functions $f_{s_1 s_2 \ldots s_{i+1}}$ and $g_{s_1 s_2 \ldots s_{i+1}}$ depend only on the following: (i) the TM M, (ii) the two functions $f_{s_1 s_2 \ldots s_i}$ and $g_{s_1 s_2 \ldots s_i}$, and (iii) the single bit s_{i+1}. In particular, the two new functions *do not depend* on remembering the string $s_1 s_2 \ldots s_i$, and this is very fortunate as arbitrarily long strings cannot be stored by a finite automaton.

At the end, when we have scanned the entire input string, we have $f_{s_1 s_2 \ldots s_n}$. We check that $f_{s_1 s_2 \ldots s_n}(q_0) = q_{\text{accept}}$, and accept iff that is the case.[5] □

EXERCISE 1.20. Explain precisely how to compute $f_{s_1 s_2 \ldots s_{i+1}}$ and $g_{s_1 s_2 \ldots s_{i+1}}$.

CLAIM 1.21. If a language L is not regular, then it requires $\Omega_{\text{weak}}(\log \log n)$ space. In other words, for any TM deciding L, there exists a constant c so that for *infinitely* many n's, there exists an input x of length n on which the machine will take at least $c \cdot \log \log n$ many steps.

PROOF. Suppose that L is not regular. Then, by claim 1.19, we know that it cannot be decided in constant space. Let M be any machine deciding L.

We want to show that there exists an infinite sequence $\{n_i\}$ (and a constant c which depends only on M) such that $\forall n_i$, $\exists x_i$ such that $|x_i| = n_i$, and M requires at least $c \cdot \log \log n_i$ space to decide x_i.

To accomplish this, we use the fact that for any k we choose, we can always find an input x, such that M requires more than k squares of space to decide x. Pick a k_1 (e.g., $k_1 = 100$—it does not matter what it is) and find an x_1 of minimal length n_1, such that M requires $s_1 \geq k_1$ space to decide x_1. We show now that $c \cdot \log \log n_1 \leq s_1$; the c will be fixed, and its value will transpire later in the proof.

[5] The original proof showing that a "one-way-automaton" can simulate a "two-way-automaton" can be found in [**She59**].

For each $j = 1, 2, \ldots, (n_1 - 1)$, let S_j be the sequence of configurations that M is in whenever its head on the input tape crosses from square j to square $j+1$ or vice-versa. We can think of S_j as a sequence of "snapshots" of M taken only when the head of the input tape crosses between squares j and $j+1$. Thus S_j is the crossing sequence associated with square j, but S_j contains more information than the crossing sequence S defined in theorem 1.11, as S_j contains the state, and the contents of the entire work-tape, and the position of the head on the work-tape.

There are at most $N = |Q| \cdot s_1 \cdot |\Gamma|^{s_1}$ possible snapshots, and there are

$$N^1 + N^2 + \cdots + N^m < (N^{m+1} - 1)/(N - 1)$$

possible crossing sequences of length at most m.

Since x_1 was chosen to be of minimal length, no two crossing sequences on x_1 are equal (for otherwise, the portion of the input between them could be eliminated, obtaining a new shorter input that still requires s_1 squares of space), so we know that $(n_1 - 1) \leq (N^{m+1} - 1)/(N - 1)$, and so $n_1 \leq N^{m+1}$. On the other hand, $m \leq N$ because otherwise we would have a loop, and M is a decider. Thus, $n_1 \leq N^{N+1}$, i.e.,

$$n_1 \leq (|Q| \cdot s_1 \cdot |\Gamma|^{s_1})^{(|Q| \cdot s_1 \cdot |\Gamma|^{s_1} + 1)}.$$

By taking log of both sides twice we get what we want; note that the constant c arises from $|Q|, |\Gamma|$, and so it depends only on M and not the size of the input.

We now pick k_2 sufficiently large so that there exists an x_2 of minimal length $n_2 > n_1$ such that M requires $s_2 \geq k_2$ space to decide x_2. Since L is not regular, we know that no matter how large k_2 is, we can always find an x_2 that requires at least k_2 space. The only problem is how to ensure that $n_2 > n_1$? As we are always picking an x_2 of minimal length (this is necessary for the argument to work) we might end up with $n_2 = n_1$. But there are finitely many x_2's of length n_1 (i.e., 2^{n_1}), so to make sure that this does not happen, we take a k_2 larger than the required space of any input of length n_1.

This procedure can be repeated *ad infinitum* to obtain $\{n_i\}$. □

If a language L can be decided in $o(\log \log n)$ space, then it is *not* the case that it requires $\Omega_{\text{weak}}(\log \log n)$ space, and so by the contrapositive of claim 1.21 it follows that it must be regular. This proves theorem 1.17. Note that this proof is not constructive, in the sense that we do not obtain a finite automaton from the $o(\log \log n)$-space bounded TM.

1.4. Answers to selected exercises

Exercise 1.1. $\{x \in \Sigma^* | \exists k \in \mathbb{N}, \exists u, v \in \Gamma^* \text{ such that } (q_{\text{init}}, \triangleright, x) \overset{M^k}{\to} (q_{\text{accept}}, u, v)\}$ and $\{x \in \Sigma^* | \exists u, v \in \Gamma^* \text{ such that } (q_{\text{init}}, \triangleright, x) \overset{M^*}{\to} (q_{\text{accept}}, u, v)\}$.

Exercise 1.5. Concatenate all the k tapes into one tape, and simulate one move of M with two passes of the entire tape of M' (the first pass of M' to determine what is underneath the heads on the tapes of M, and the second pass to make the necessary changes). Note that M uses at most $f(n)$ squares of each of its tapes. This is an

observation that we make all the time: in t many steps, a TM can write on at most the first t squares.

Exercise 1.7. The idea is to "increase the word size." Introduce new symbols which encode several symbols of M.

Exercise 1.8. It only says that if a language can be decided in time $O(n)$, then it can be decided in time $(1+\varepsilon)n+2$, for any $\varepsilon > 0$, and hence in a time that is almost strictly linear. That is, the constant can be made arbitrarily close to 1. As far as "big-Oh" notation, it is a way of justifying it, as the theorem says that constants are not important.

Exercise 1.9. At least not with the proof given, since it depends crucially on increasing the word size.

Exercise 1.13. We showed that there exists a constant c so that for every n there exists an $x_0 \in \{0,1\}^n$, so that M takes at least $c \cdot n^2$ many steps to decide $x_0 0^n x_0^R$. So technically what we showed is that among inputs of length $3 \cdot n$ (where n is sufficiently big) there exists at least one input on which M takes time $c \cdot n^2$; or, equivalently, given inputs of length n, where n is sufficiently big *and* divisible by 3, there exists at least one such input on which M takes $c \cdot \left(\frac{n}{3}\right)^2 = \left(\frac{c}{9}\right) \cdot n^2$ steps. So, what we really showed is that L_{pal} requires $\Omega_{\text{weak}}(n^2)$ steps to be decided on a single tape TM—see §1.3.2 for a definition of Ω_{weak}. So in the statement of theorem 1.11 we should replace $\Omega(n^2)$ by $\Omega_{\text{weak}}(n^2)$.

Exercise 1.16. L_{pal} is in linear-time because we can copy the input string w to the second tape, move the second head to the end, and then move the two heads simultaneously towards each other, comparing symbols. It is in logspace because we can work in $|w|$ many stages, in stage i we compare w_i to $w_{|w|+1-i}$. We keep track of the stages with one counter ($O(\log |w|)$ many bits, and in each stage compute the positions i and $|w|+1-i$ with a second counter of the same size.

Exercise 1.18. Based on exercise 4, homework 2, in [**Koz06**]; the solution can be found on page 324 of the same book.

Exercise 1.20. Suppose we already have $g_{s_1 s_2 \ldots s_{i+1}}$. Then,

$$f_{s_1 s_2 \ldots s_{i+1}}(q) = g_{s_1 s_2 \ldots s_{i+1}}(f_{s_1 s_2 \ldots s_i}(q)).$$

We obtain $g_{s_1 s_2 \ldots s_{i+1}}(q)$ as follows: if the head (when started on s_{i+1}) moves right, we are done—we simply take $g_{s_1 s_2 \ldots s_{i+1}}(q)$ to be the new state of the machine. But if the head moves left and enters a state q', we must run $g_{s_1 s_2 \ldots s_i}(q')$, and repeat. How can we ensure that this procedure ends? That is, how can we ensure that the head will eventually move right? In the proof of claim 1.19 we are transforming a machine M into a new machine M' where the head moves from left-to-right only. In order to ensure that the transformation works properly M must scan the *entire* input before accepting—this forces the head to move right eventually. (Of course, we can force M to scan the entire input by insisting that it goes all the way to the end of the string before accepting.)

1.5. Notes

Exercise 1.3 is based on the presentation of URIM machines in [**Coo08**]. For a detailed proof of theorem 1.4 see [**Pap94**, theorem 2.1], for theorem 1.6 see [**Pap94**, theorem 2.2], and for theorem 1.10 see [**Pap94**, theorem 2.6]. §1.3.1 is based on [**Pap94**, exercise 2.8.5], and it is also presented in [**Koz06**].

An interesting question is what is the smallest possible number of states and symbols necessary to be able to define a UTM. Minsky showed in [**Min62**] that a UTM can be constructed with seven states and a tape alphabet of four symbols. A recent breakthrough seems to suggest that we can get away with two states and three symbols (see the Wolfram Blog on Alex Smith). The *Busy beaver* function $\Sigma(n)$ is related to this question. The value of $\Sigma(n)$ is the largest number of 1s that a TM may print on its tape, when started on the empty tape, having an alphabet of two symbols, $\{0, 1\}$, and using n states. $\Sigma(n)$ is not computable.

Following the historical remarks in [**BM77**, §17], we point out that imaginary computers in the style of URIMs were invented by several people independently (Shepherdson and Sturgis, Lambek, Minsky) in the late 1950s. The first definition of the class of recursive functions was given by Gödel in 1934 following a suggestion by Herbrand. This definition was proved by Kleene in 1936 to be equivalent to the definition which we are presenting here.

An even more ambitious problem than exercise 1.3 would be to show that the Turing machine definition of recursion is equivalent to the ZFC (Zermelo-Fraenkel set theory with the Axiom of Choice) definition of recursion. This definition is given as follows: take the language $\mathcal{L} = \{\in, =\}$, i.e., the language consisting of two binary predicates. We say that the set $S \subseteq \mathbb{N}$ is recursive if there exists a first order formula α over the language $\mathcal{L}$, such that α has a single free variable x and if $m \in S$ then $\mathrm{ZFC} \vdash \alpha(\bar{m})$, and if $m \notin S$ then $\mathrm{ZFC} \vdash \neg\alpha(\bar{m})$. Here $\bar{m}$ is the representation of the ordinal m in the language $\mathcal{L}$. Of course, it would be easier to do this in PA (Peano Arithmetic)—which is already sufficient for the definition of recursion. All these different, but equivalent, definitions of recursion should be convincing evidence that we have captured well the notion of "computable."

The invention of palindromes is generally attributed to Sotades the Obscence of Maronea, who lived in the third century BC in Greek-dominated Egypt. Surprisingly, palindromes appear not just in witty word games (such as `madamimadam` in James Joyce's *Ulysses*, or the title of the famous NOVA program, *A Man, a Plan, a Canal, Panama*), but also in the structure of the male defining chromosome. Other human chromosome pairs fight damaging mutations by swapping genes, but because the Y chromosome lacks a partner, genome biologists have previously estimated that its genetic cargo was about to dwindle away in perhaps as little as five million years. However, researchers on the sequencing team discovered that the chromosome fights withering with palindromes. About six million of its fifty million DNA letters form palindromic sequences—sequences that read the same forward as backward on the two strands of the double helix. These copies provide backups in case of a bad mutation. (These observations come from [**Liv05**].)